



Guide de codage des programmes automates

Itris Automation Square

20 janvier 2012 - Version 1.6.7

<http://www.automationsquare.com>

Contents

1	Objectif de ce document	5
2	Structure de ce document	6
2.1	Classification des règles	6
2.2	Présentation de chaque règle	6
3	Règles de codage	7
3.1	Nommage	7
3.1.1	Règles communes à tous les automates	7
3.2	Commentaires	9
3.2.1	Règles communes à tous les automates	9
3.3	Ecriture du code	10
3.3.1	Règles communes à tous les automates	10
3.3.2	Règles spécifiques à Schneider Electric Unity	11
3.4	Structuration	11
3.4.1	Règles communes à tous les automates	11
3.5	Informations utiles	13
3.5.1	Informations utiles communes à tous les automates	13
3.6	Options	15
3.6.1	Options Unity des projets automates	15
4	Annexes	18
4.1	Résumé des règles	18
4.2	Historique du document	20

Chapter 1

Objectif de ce document

Ce document propose un style de codage pour la programmation des automates industriels, afin d'améliorer la lisibilité, la maintenabilité et la cohérence des programmes. Il décrit des règles simples permettant d'obtenir, dès leur première application, des résultats probants sur la qualité du code automate.

Toutes les règles présentes dans ce document sont vérifiables automatiquement à l'aide de l'outil PLC Checker et ce pour les automates programmés avec les ateliers PL7-PRO et Unity PRO de Schneider Electric, STEP7 de Siemens et RSLogix 5000 de Rockwell Automation.

Ce document a été rédigé à partir d'une analyse critique de nombreux référentiels déjà utilisés par les clients d'Itris Automation Square dans des domaines d'activité variés et possédant des gammes d'automates diversifiées. Il s'inspire également de l'expérience des équipes d'automaticiens d'Itris Automation Square.

Chapter 2

Structure de ce document

2.1 Classification des règles

Les règles contenues dans ce document sont classées dans plusieurs catégories : Nommage, Commentaire, Ecriture, Structuration. Vient ensuite une partie appelée "Informations utiles" qui propose également des pistes de réflexion. Elles ne constituent pas à proprement parler des règles de codage, mais peuvent permettre l'amélioration de la qualité des programmes. Pour finir des règles de vérifications des options du projet sont proposées pour certains ateliers.

Chaque catégorie est éventuellement subdivisée en deux parties : les règles qui peuvent s'appliquer indépendamment de l'automate pour lequel le programme est développé et celles qui ne sont pertinentes que pour un automate particulier ou qui nécessitent d'être déclinées afin d'être effectivement applicables.

2.2 Présentation de chaque règle

Les règles sont numérotées en accord avec leur implémentation dans PLC Checker, l'outil de vérification automatique des règles de codage proposé par Itris Automation Square. Pour chaque règle, on retrouve une structure identique composée d'un titre de section qui résume la règle, de la règle proprement dite et de commentaires justifiant notamment la présence de la règle dans le document. Lorsque cela semblait nécessaire pour faciliter la compréhension de la règle, un exemple de code a été rédigé.

Il est à noter que les explications complémentaires et les exemples n'ont pas pour but de constituer une formation exhaustive à l'aspect de la programmation auquel il est fait référence dans la règle.

Chapter 3

Règles de codage

3.1 Nommage

Les éléments manipulés dans les ateliers automates portent des mnémoniques. Ce sont par exemple, des variables, des routines (sections, sous-routines, FC...), des blocs fonction (FB). L'objectif de cette section est de s'assurer que les mnémoniques suivent des règles assurant la lisibilité, la maintenabilité et permettant une plus grande pérennité du code. D'une manière générale, les éléments propres aux librairies des ateliers constructeurs sont exclus des contrôles de nommage. Bien entendu, lorsque les ateliers le permettent, nous déconseillons de modifier les noms des éléments propres aux librairies.

3.1.1 Règles communes à tous les automates

N.1 - Tous les éléments constituant le programme doivent être nommés

Tous les éléments constituant le programme doivent être nommés (Programme, Modules, Variables, DFB, FB, OB, SR).

Commentaire : Si tous les éléments du programme sont nommés de façon significative, il est plus facile de lire et de comprendre le programme ; la maintenance en devient facilitée. Par ailleurs, ne pas utiliser de mnémonique revient à lier le code à une implémentation matérielle donnée, ce qui induit une perte d'évolutivité.

N.2 - Les noms des éléments du programme doivent avoir une taille minimum et maximum

Les noms des éléments du programme doivent comporter au moins 4 caractères et au plus 30 caractères.

Commentaire : Un nombre minimum de caractères assure un nommage plus significatif. Un maximum de caractères permet :

- sous Unity, PL7pro et Rockwell, d'éviter les mnémoniques tronqués à l'affichage, à l'export, ...
- sous Step7, d'améliorer la lisibilité

N.3 - Les mnémoniques des éléments ne font pas référence à leur adresse physique

Les mnémoniques des éléments ne doivent pas contenir une référence à leur adresse physique (par exemple, "MW2", "FB4"...).

Commentaire : La référence à l'adresse physique est susceptible de ne plus être pertinente suite à une évolution du code et donc de nuire à la lisibilité du code voire d'induire une confusion dangereuse. Il est notable que, du fait de l'évolution des ateliers logiciels, la notion de couplage avec l'adresse physique n'est pas nécessairement pertinente pour certains automates pour lesquels les variables ne sont pas localisées. Une telle référence à l'adresse physique peut être involontaire, elle n'en demeure pas moins interdite.

Exemple :

Adresse Physique	Mnémoniques	Commentaire
%MW23	Cde_23	Ne jamais lier le nommage de la variable à son adresse physique

N.4 - Les mnémoniques de mise au point ne doivent pas faire partie du programme final

La mise au point ne doit pas faire partie du programme. Notamment, certains noms d'éléments (*test, toto, titi, tata, tutu, a effacer, todo...*) sont interdits et les commentaires ne doivent pas contenir d'expressions laissant penser que le programme est encore en cours d'écriture (*a faire plus tard, a effacer, todo*)

Commentaire : Laisser des éléments liés à la mise au point dans le programme en production nuit à sa lisibilité et peut avoir pour conséquences un comportement différent de celui attendu. La liste d'éléments interdits est évolutive.

N.5 - Les caractères spéciaux sont interdits dans les noms des mnémoniques

Les caractères spéciaux sont interdits dans les noms des mnémoniques. Seuls les caractères alphanumériques et le "souligné" (*underscore*) sont autorisés. En particulier, l'espace et les accents sont interdits.

Commentaire : Utiliser des caractères spéciaux dans les noms des mnémoniques nuit à la portabilité du code vers des versions postérieures potentielles des ateliers de développement et/ou vers d'autres plateformes. Ils nuisent aussi à la lisibilité dans un contexte international. Enfin, restreindre le jeu de caractères autorisés permet de s'assurer d'une meilleure distinction entre les noms des variables.

Exemple :

```
départ := true;

# .... beaucoup plus loin dans le code

depart := false;      # Le fait d'autoriser le caractère spécial 'é' conduit à une confusion
possible entre les mnémoniques.
```

N.6 - Les mots clefs des langages de programmation ne doivent pas servir de mnémoniques

Les mots clefs des langages de programmation (par exemple "if", "then", "else", "while", "for", "repeat", "until", "begin", "end", "do", "function", "procedure", "exit", "wait", "start", "stop", "return",...) ne peuvent pas être utilisés comme noms des éléments du programme. En revanche, les mnémoniques peuvent contenir un mot clé du langage.

Commentaire : Utiliser des mots clefs des langages comme noms des mnémoniques nuit à la lisibilité, à la portabilité du programme et peut poser des problèmes lors de la compilation du code. Il est à noter qu'un certain nombre d'ateliers interdisent déjà un tel nommage.

Exemple :

```
start := true;      # Mauvais mnémonique, start est un mot-clé du langage
start_moteur := true; # Correct mnémonique
```

3.2 Commentaires

Au delà du nommage, il est important de suivre des règles quant à la bonne utilisation des commentaires. En effet, plus un commentaire est explicite et détaillé, plus il facilitera la compréhension du code. D'une manière générale, les éléments propres aux bibliothèques des ateliers constructeurs sont exclus des contrôles de commentaires.

3.2.1 Règles communes à tous les automatés

C.1 - Tous les éléments constituant le programme doivent être commentés

Tous les éléments constituant le programme doivent être commentés (Programme, Modules, Variables, DFB, FB, OB).

Commentaire : Si tous les éléments du programme sont commentés de façon significative, le programme est plus facile à lire donc plus facile à comprendre et donc plus facile à maintenir.

C.2 - Les commentaires doivent comporter un minimum de caractères

Les commentaires doivent comporter un minimum de caractères (7 pour les variables, 15 pour les codes) pour éviter les commentaires non significatifs.

Commentaire : Un nombre minimum de caractères assure que les commentaires sont plus significatifs.

C.3 - Chaque réseau doit être commenté

Lorsqu'ils sont utilisés, les réseaux (aussi appelés rungs) doivent être commentés. Cette règle est spécifique à certains langages de la norme IEC61131. Par ailleurs, la notion de rung n'est pas toujours présente dans les ateliers de développement des constructeurs d'automates.

Commentaire : Au delà du taux de commentaire global, il est important d'assurer une bonne répartition des commentaires. Une telle règle est aussi un guide qui assure le respect d'une certaine cohérence dans le découpage du code.

C.4 - Les commentaires ne contiennent pas le marqueur de début de commentaire

Les commentaires ne doivent pas contenir le marqueur de début de commentaire (par exemple, `/* */`, `/*(*/`).

Commentaire : La présence d'un marqueur de début de commentaire dans un commentaire est souvent le symptôme d'un oubli de marqueur de fin de commentaire, pouvant conduire à la non-exécution d'une partie de code.

Exemple:

```
/* Un commentaire quelconque dont le marqueur de fin a été oublié
```

```
Section_critique_qu_il_faut_a_tout_prix_executer_mais_qui_se_retrouve_commentee();  
/* <-Ce marqueur de début de commentaire est en commentaire car le marqueur de fin du commentaire  
précédent a été oublié */
```

3.3 Ecriture du code

3.3.1 Règles communes à tous les automatés

E.1 - Toutes les variables doivent être écrites avant d'être lues

A l'exception des entrées et des variables système, toutes les variables doivent être écrites avant d'être lues durant un cycle automate.

Commentaire : Une variable lue avant d'être écrite a pour conséquence de rendre le code non-réactif : un code efficace doit respecter un cycle lecture des entrées → traitement → mise à jour des sorties. Par ailleurs, il faut s'assurer que les variables qui ne sont jamais écrites par le code ont bien été initialisées ou proviennent de l'extérieur du programme.

Dans certain cas, il est justifié de ne pas respecter cette règle :

- pour les variables qui ont été initialisées dans la base de données,
- pour les variables en provenance des communications,
- pour les variables d'état qui mémorisent la valeur d'un cycle antérieur.

Exemple:

```
if alarmeUrgente then
  traitementAlarme;
end if;

# ...

alarmeUrgente := presenceDefault AND ...;

# Dans ce cas là, l'appel de la routine de traitement d'alarme se fait lors du cycle suivant
# ce qui augmente le temps de réponse moyen.
# Ce retard est de l'ordre de grandeur d'un tour de cycle.
```

E.2 - Les paramètres des blocs fonctionnels utilisateur doivent être utilisés correctement

Cette règle se subdivise en sous-règles :

- **E.2.a et b** - Les entrées des blocs fonctionnels utilisateur doivent être lues et ne pas être écrites.
- **E.2.c et d** - Les entrées/sorties des blocs fonctionnels utilisateur doivent être lues et écrites.
- **E.2.e** - Les sorties des blocs fonctionnels utilisateur doivent être écrites avant leur éventuelle lecture.

Commentaire : Des paramètres non-utilisés de façon correcte sont des symptômes d'un code non-finalisé ou d'un code qui contient encore des éléments inutiles à son bon fonctionnement. Sur certains automatés, la mauvaise utilisation des paramètres peut générer des effets de bords potentiellement dangereux tels qu'une corruption de données. La vérification de l'utilisation des variables privées des blocs fonctionnels utilisateur se fait par la règle S7.

3.3.2 Règles spécifiques à Schneider Electric Unity

E.U.1 - L'utilisation de fonctions des bibliothèques obsolètes d'Unity est à éviter

Les fonctions des bibliothèques obsolètes ne doivent pas être utilisés dans le cadre de nouveaux projets.

Commentaire : Sous l'atelier Unity, certaines bibliothèques sont obsolètes et ont pour unique vocation d'assurer la compatibilité ascendante avec des versions antérieures des automates Schneider. L'utilisation des fonctions de ces bibliothèques est à éviter pour des problèmes de portabilité futur. De plus, certaines d'entre elles peuvent rendre le code non-déterministe (ex: les décalages sur les entiers signés: shl_int, shlz_int, shr_int, shrz_int, rol_int, ror_int, ...).

3.4 Structuration

3.4.1 Règles communes à tous les automates

S.1 - Les sauts en arrière sont interdits

Les sauts en arrière sont interdits.

Commentaire : L'utilisation d'un saut en arrière fait courir le risque de déclencher une boucle infinie, ce qui peut conduire à un arrêt de l'automate. Par ailleurs, de manière générale, l'utilisation de saut nuit à une bonne compréhension du code et donc à sa facilité de maintenance.

Exemple:

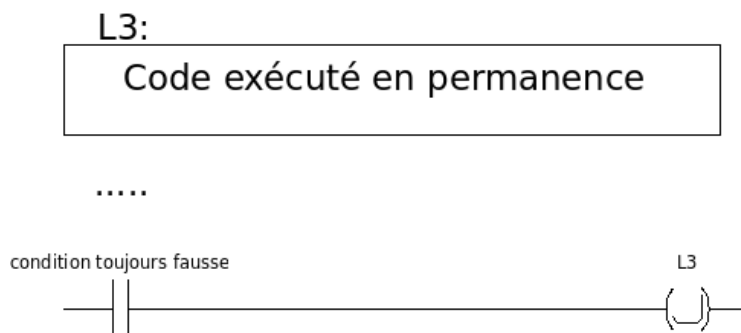


Figure 3.1: Exemple de saut en arrière provoquant une boucle infinie

S.2 - Une variable doit être écrite au sein d'une seule routine

Une variable doit être écrite au sein d'une seule routine.

Commentaire : Le respect d'une structuration simple et claire facilite la compréhension du code et donc sa maintenance. Par ailleurs, des élaborations multiples d'une variable peuvent avoir pour conséquence un comportement du code erratique.

S.3 - Une variable doit être écrite au sein d'une seule tâche

Une variable doit être écrite au sein d'une seule tâche.

Commentaire : Élaborer une variable depuis plusieurs tâches conduit à un comportement non-déterministe donc à un conflit d'accès potentiel.

S.4 - Une sortie physique ne doit être écrite qu'une seule fois par cycle automate

Une sortie physique ne doit être écrite qu'une seule fois par tour de cycle automate.

Commentaire : L'écriture multiple d'une sortie peut conduire à des problèmes de sécurité de fonctionnement. Par ailleurs, un code dans lequel les sorties sont écrites plusieurs fois est moins facile à comprendre et donc à maintenir. A noter que l'écriture dans une boucle est considérée comme une écriture multiple et constitue donc une violation de cette règle.

Exemple:

```
if condition_tres_complique then
    ma_sortie_critique := TRUE;
end if;

#... beaucoup plus loin dans le code

if une_autre_condition_tres_complique then
    ma_sortie_critique := FALSE;
end if;

# Dans le cas où cette sortie n'est pas commandée alors qu'elle le devrait, il est très
# difficile d'identifier quelle affectation est exécutée.
```

S.5 - Les variables ne doivent pas en général être localisées

La localisation des variables doit être strictement réservée aux variables dont l'élaboration ou la consommation sont extérieures au programme : variables de communication, variables d'entrées-sorties, variables de configuration de cartes spéciales, variables de tests. En revanche toutes les variables internes du programme doivent être non localisées.

Commentaire : Localiser une variable réduit les possibilités d'évolution à cause d'une organisation mémoire plus rigide. Cela réduit également la portabilité, puisque les possibilités d'organisation mémoire varient d'un automate à l'autre. Cela empêche la ré-utilisabilité du code. En effet, dans le cadre d'une autre application, la zone mémoire choisie peut être réservée pour d'autres usages. Enfin, cela peut conduire à des erreurs d'organisation mémoire dans lesquelles plusieurs variables se trouvent localisées à la même case mémoire.

S.6 - Les instances de FB/DFB sont appelées une et une seule fois

Chaque instance de bloc fonctionnel doit être appelé une fois et une seule par tour de cycle automate.

Commentaire : Une instance de bloc fonctionnel possède ses propres variables qui permettent la sauvegarde de l'état de l'instance entre deux tours de cycle. Chaque exécution de l'instance doit faire évoluer ces variables pour rendre compte du changement d'état à ce tour de cycle. Une exécution multiple pour la même instance de ce code semble faire avancer les cycles plus vite que la réalité pour cette instance. En fonction du programme, cela peut conduire à des erreurs difficiles à diagnostiquer.

Enfin, une même instance peut être appelée deux fois suite à un copier/coller pas finit, dans lequel les paramètres ont été modifiés mais pas le nom de l'instance.

S.7 - Les variables définies (hors réserves) sont utilisées

Les variables déclarées doivent être lues ou écrites à l'exception des variables qui ont été définies comme variables de réserve.

Commentaire : Les variables déclarées et non utilisées sont souvent le signe d'une base de données obsolète ou pas à jour qui rend le travail de la maintenance plus compliqué.

S.8 - Les variables ne se chevauchent pas

Dans le cas où des variables doivent être localisées, deux variables ne doivent pas utiliser la même localisation.

Commentaire : Les variables localisées qui se chevauchent sont équivalentes à un transfert implicite de valeur entre elles. Cela peut nuire à la lisibilité du code. C'est parfois un signe d'une mauvaise organisation mémoire qui peut rester invisible lors des tests et conduire à un comportement non déterministe du programme lors de l'exploitation.

S.9 - Mesure de la complexité

Le code complexe cache des erreurs.

Commentaire : Certaines parties de programmes sont plus complexes que d'autres. C'est particulièrement vrai lorsque le flot d'exécution séquentiel du programme est interrompu. Cette règle recherche certaines causes d'interruption du flot que sont les boucles, les sauts arrière et les niveaux d'imbrications d'instructions complexes trop élevés.

S.10 - Limitations dues aux SCADA

Certains logiciels de supervision ne supportent pas les tableaux de structure. Cette règle les identifie.

Commentaire : Certains logiciels SCADA ne supportent pas les tableaux de structures. Le développeur doit donc vérifier que les tableaux de structure ne sont pas utilisés pour être échangés avec la supervision.

3.5 Informations utiles

3.5.1 Informations utiles communes à tous les automatés

I.1 - Taux de commentaires

Il est recommandé que le code contienne un pourcentage minimum de commentaires.

Commentaire : A titre indicatif, Itris Automation Square recommande un taux de 30% (nombre d'instructions commentées/nombre total d'instructions). Un tel taux minimum de commentaires assure que le développement a été réalisé en ayant à l'esprit un souci de maintenabilité et de bonne compréhension par des intervenants postérieurs.

I.2 - Présence de code dans les commentaires

Il est important de vérifier la présence éventuelle de code sous forme de commentaires.

Commentaire : La présence de code dans des commentaires est souvent due à la mise en commentaire d'une partie du code dans le cadre des phases de test et de debug. Le code de production ne doit pas contenir de traces des phases de mise au point.

I.3 - Présence de code mort

Il faut évaluer la présence éventuelle de code mort dans le programme.

Commentaire : Le code mort nuit à la lisibilité et la maintenabilité du programme et peut être le symptôme d'un problème fonctionnel. Attention, si PLC Checker détecte les fonctions non-référencées et les parties de code isolées derrière un saut incondtionnel, aucune exhaustivité de détection n'est cependant garantie (comme, par exemple, pour le code mort lié à l'évaluation d'une expression).

I.4 - Liste des éléments verrouillés

Les éléments verrouillés ne sont pas analysés par PLC Checker, il convient donc de les identifier.

Commentaire : Sur des applications où la majeure partie du code est verrouillée, il est important de le savoir.

I.5 - Nombre cyclomatique $v(G)$

Le nombre cyclomatique donne une information relative à la complexité d'une routine. Il est important que le nombre cyclomatique reste raisonnablement bas.

Commentaire : Le nombre cyclomatique représente le nombre de chemins d'exécution différents qui peuvent être exécutés dans une routine. Plus ce nombre est élevé, plus il est difficile de valider le comportement correct de cette routine. Il est alors nécessaire de diviser celle-ci en d'autres sous-routines de nombre cyclomatique inférieur au seuil.

I.6 - Complexité essentielle $ev(G)$

La complexité essentielle donne une information relative à l'utilisation d'instructions non structurées. Il est important que la complexité essentielle reste raisonnablement bas.

Commentaire : La complexité essentielle représente le nombre d'instructions non structurées appartenant à une routine. Plus ce nombre est élevé, plus il est difficile de valider le comportement correct de cette routine. La plupart du temps il est possible de remplacer des instructions non structurées par des instructions structurées.

I.7 - Nombre de lignes de code

Le nombre de lignes de code d'une procédure ou d'un bloc fonction est une métrique très simple qu'il convient de garder sous contrôle.

Commentaire : De très grosses procédures ou blocs fonctions ont de mauvaises propriétés de test, validation, lisibilité et maintenance.

I.8 - Nombre d'étapes du Grafctet

Il s'agit du nombre d'étapes Grafctet présentes.

Commentaire : Cette règle retourne le nombre d'étapes trouvées dans un Grafctet de l'application. Ce nombre peut servir à détecter des Grafctets plus complexes parmi tous les Grafctets d'une application.

I.9 - Nombre de branches du Grafctet

Le nombre est la somme du nombre de branches de chacune des divergences ET/OU trouvées dans un Grafctet donné.

Commentaire : Cette règle calcule le nombre de branches d'un Grafctet. Ce nombre donne des informations sur la complexité d'un Grafctet donné. Ce nombre est équivalent à la complexité cyclomatique ($v(G)$) calculé pour les langages de programmation impératifs.

I.10 - Ratio de code dupliqué sur l'application

Ce nombre détecte le pourcentage de code qui est dupliqué.

Commentaire : Réutiliser le code en le copiant/collant n'est pas une bonne pratique. La maintenance et l'évolution du programme deviennent plus difficiles car les modifications doivent être appliquées à plusieurs endroits. De plus l'opération manuelle consistant à copier/coller est souvent responsable d'erreurs difficiles à détecter : la phase de modification qui suit le copier/coller est parfois incomplète.

3.6 Options

3.6.1 Options Unity des projets automates

O.U.1 - SFC : Le multi-jetons sur le Grafcet est interdit

Sous l'atelier Unity, dans les options du projet, il faut décocher la possibilité de faire tourner le Grafcet en mode multi-jetons.

Commentaire : Le mode multi-jetons sous Unity n'a que pour vocation d'assurer la compatibilité ascendante avec des versions antérieures des automates Schneider. Il ne doit pas être utilisé dans le cadre de nouveaux projets car il peut rendre le code non-déterministe.

Exemple: Si le code Grafcet suivant est exécuté avec une condition A toujours vraie, après plusieurs tours de cycles, toutes les étapes sont actives simultanément.

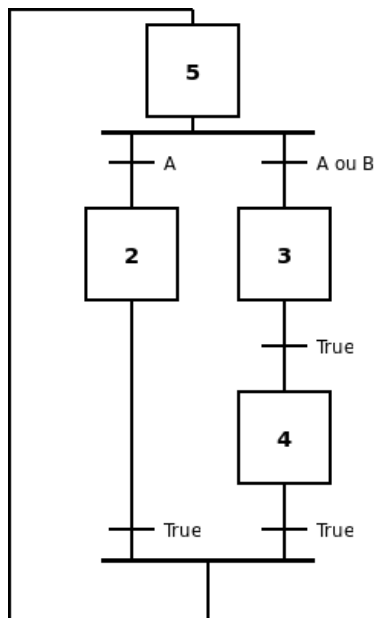


Figure 3.2: Grafcet non déterministe en mode multi-jeton

O.U.2 - SFC : Non maintien des étapes précédentes actives pour SetSteps

Sous l'atelier Unity, dans les options du projet, il faut décocher l'option « SetSteps : maintien des étapes précédentes actives ».

Commentaire : Cette option est liée au mode multi-jetons - voir O.U.1.

O.U.3 - SFC : Pas de saut In/Out dans les divergences en ET

Sous l'atelier Unity, dans les options du projet, il faut décocher l'option « Divergence en ET : autoriser saut In/Out ».

Commentaire : Cette option est liée au mode multi-jetons - voir O.U.1.

O.U.4 - SFC : Une seule évolution par divergence

Sous l'atelier Unity, dans les options du projet, il faut décocher l'option « Autoriser plusieurs évolutions par divergence ».

Commentaire : Cette option est liée au mode multi-jetons - voir O.U.1.

O.U.5 - ST : Sauts et labels interdits

Sous l'atelier Unity, dans les options du projet, il faut décocher l'option « Autoriser les sauts et les labels ».

Commentaire : De manière générale, l'utilisation de saut nuit à une bonne compréhension du code et donc à sa facilité de maintenance.

O.U.6 - Pas de chiffre en début d'identificateur

Sous l'atelier Unity, dans les options du projet, il faut décocher l'option « Chiffres en début autorisés ».

Commentaire : De manière générale, l'utilisation de chiffres en début de mnémoniques nuit à la lisibilité et à la portabilité du programme.

O.U.7 - Jeu de caractères standard pour les identificateurs

Sous l'atelier Unity, dans les options du projet, il faut sélectionner le jeu de caractères Standard (et pas Etendu ou Unicode) pour les identificateurs.

Commentaire : De manière générale, l'utilisation d'un jeu de caractères non standard nuit à la lisibilité et à la portabilité du programme.

Exemple : Il peut, par exemple, être difficile de faire la différence entre une variable "départ" et une autre variable "depart"

O.U.8 - Les expressions ST sont interdites dans le FBD et le LD

Sous l'atelier Unity, dans les options du projet, il faut décocher l'option « Utilisation d'expressions ST ».

Commentaire : De manière générale, l'utilisation d'expression ST dans les langages FBD et LD nuit à la lisibilité du programme.

O.U.9 - Les informations d'upload avec commentaires et tables d'animation

Sous l'atelier Unity, dans les options du projet, il faut :

1. cocher l'option « Avec » les informations d'upload,
2. cocher l'option « Commentaires (variables et types) » pour les informations d'upload,
3. cocher l'option « Tables d'animation » pour les informations d'upload

Commentaire : Dans la mesure où la mémoire automate le permet, le stockage des informations d'upload le plus complet possible permet de retrouver ces informations en cas de perte de la sauvegarde du programme, ce qui améliore la maintenabilité à long terme.

O.U.10 - Les bobines en LD sont alignées à droite

Sous l'atelier Unity, dans les options du projet, il faut cocher l'option « Bobines alignées à droite » pour l'éditeur Ladder.

Commentaire : L'alignement des bobines sur la droite de l'éditeur permet une meilleure lisibilité du programme.

Chapter 4

Annexes

4.1 Résumé des règles

Cette annexe récapitule toutes les règles et informations utiles mentionnées dans ce document.

Règles de Nommage

- N.1 - Non-spécifique - Tous les éléments constituant le programme doivent être nommés
- N.2 - Non-spécifique - Les noms des éléments du programme doivent avoir une taille minimum et maximum
- N.3 - Non-spécifique - Les mnémoniques des éléments ne font pas référence à leur adresse physique
- N.4 - Non-spécifique - Les mnémoniques de mise au point ne doivent pas faire partie du programme final
- N.5 - Non-spécifique - Les caractères spéciaux sont interdits dans les noms des mnémoniques
- N.6 - Non-spécifique - Les mots clefs des langages de programmation ne doivent pas servir de mnémoniques

Règles sur les Commentaires

- C.1 - Non-spécifique - Tous les éléments constituant le programme doivent être commentés
- C.2 - Non-spécifique - Les commentaires doivent comporter un minimum de caractères
- C.3 - Non-spécifique - Chaque réseau doit être commenté
- C.4 - Non-spécifique - Les commentaires ne contiennent pas le marqueur de début de commentaire

Règles d'écriture

- E.1 - Non-spécifique - Toutes les variables doivent être écrites avant d'être lues
- E.2.a, E.2.b - Non-spécifique - Les entrées des blocs fonctionnels utilisateur doivent être lues et ne pas être écrites
- E.2.c, E.2.d - Non-spécifique - Les entrées/sorties des blocs fonctionnels utilisateur doivent être lues et écrites
- E.2.e - Non-spécifique - Les sorties des blocs fonctionnels utilisateur doivent être écrites avant leur éventuelle lecture
- E.U.1 - SE Unity - L'utilisation de fonctions des bibliothèques obsolètes d'Unity est à éviter

Règles de Structuration

- S.1 - Non-spécifique - Les sauts en arrière sont interdits
- S.2 - Non-spécifique - Une variable doit être écrite au sein d'une seule routine
- S.3 - Non-spécifique - Une variable doit être écrite au sein d'une seule tâche
- S.4 - Non-spécifique - Une sortie physique ne doit être écrite qu'une seule fois par cycle automate
- S.5 - SE Unity - Les variables ne doivent pas en général être localisées
- S.6 - Non-spécifique - Les instances de FB/DFB sont appelées une et une seule fois
- S.7 - Non-spécifique - Les variables définies sont utilisées
- S.8 - Non-spécifique - Les variables ne se chevauchent pas
- S.9 - Non-spécifique - Mesure de la complexité
- S.10 - Non-spécifique - Limitations dues aux SCADA

Informations Utiles

- I.1 - Non-spécifique - Taux de commentaires
- I.2 - Non-spécifique - Présence de code dans les commentaires
- I.3 - Non-spécifique - Présence de code mort
- I.4 - Non-spécifique - Présence d'éléments verrouillés
- I.5 - Non-spécifique - Nombre cyclomatique $v(G)$
- I.6 - Non-spécifique - Complexité essentielle $ev(G)$
- I.7 - Non-spécifique - Nombre de lignes de code
- I.8 - Non-spécifique - Nombre d'étapes du Grafcet
- I.9 - Non-spécifique - Nombre de branches du Grafcet
- I.10 - Non-spécifique - Ratio de code dupliqué sur l'application

Options

- O.U.1 - SE Unity - SFC : Le multi-jetons sur le Grafcet est interdit
- O.U.2 - SE Unity - SFC : Non maintient des étapes précédentes actives pour SetSteps
- O.U.3 - SE Unity - SFC : Pas de saut In/Out dans les divergences en ET
- O.U.4 - SE Unity - SFC : Une seule évolution par divergence
- O.U.5 - SE Unity - ST : Sauts et labels interdits
- O.U.6 - SE Unity - Pas de chiffre en début d'identificateur
- O.U.7 - SE Unity - Jeu de caractères standard pour les identificateurs
- O.U.8 - SE Unity - Les expressions ST sont interdites dans le FBD et le LD
- O.U.9 - SE Unity - Les informations d'upload avec commentaires et tables d'animation
- O.U.10 - SE Unity - Les bobines en LD sont alignées à droite

4.2 Historique du document

Date	Changements
9 juin 2009	Version initiale
26 octobre 2010	Ajout de la section sur les options
9 novembre 2011	Ajout les descriptions pour les règles S5, S6, S7, S8, S9, S10, I5, I6, I7, I8, I9, I10
21 novembre 2011	Corrections texte dans Annexes et C4
19 janvier 2012	Relecture et corrections (ortho, gramm) par Pauline